INDEX

```
               ########    ####    ###   ##     ##    ##
                  ##        ##     ####  ##      ##   ##
                  ##        ##     ## ## ##       ####
                  ##        ##     ##  ####        ##
                  ##       ####    ##   ###        ##


          #######        ##        ######     ####     ######
          ##    ##      ####      ##           ##     ##    ##
          #######      ##  ##      ######      ##     ##
          ##    ##    ########          ##     ##     ##    ##
          #######    ##      ##    ######     ####     ######
```

E X P E R I M E N T E R ' S   K I T

Copyright (C) 1977 by Tom Pittman

GETTING THE MOST OUT OF TINY BASIC

TINY BASIC in the 6800 and 6502 was designed to be a small but powerful language for hobbyists. It allows the user to write and debug quite a variety of programs in a language more "natural" than hexadecimal absolute, and programs written in TINY are reasonably compact. Because the language is small it is not as convenient for some applications as perhaps a larger BASIC might be, but the enterprising programmer will find that there is very little that cannot be done from TINY with only occasional recourse to machine language. This is, in fact, as it should be: The high level language provides the framework for the whole program, and the individual esoteric functions done in machine language fill in the gaps.

For the remainder of this article we will assume one of the standard TINY BASIC programs which follow the memory allocations defined in Appendix D of the User Manual[1]. Specifically, memory locations 0020-0023 (hex) contain the boundaries of the user work space, and so on. If your system differs from this norm, you may have to make adjustments to Page 00 address locations referenced here, but everything else should be applicable. Because there are almost as many different starting addresses for the TINY BASIC code as there are versions, we will assume that the variable "S" contains the starting address. In other words, for the "R" version (Mikbug) S=256, the "K" and "S" versions S=512, for "T" (KIM-2 4K) S=8192, etc.

THE USR FUNCTION

Perhaps the least understood feature of TINY BASIC is the machine language subroutine call facility. Not only is it useful for calling your own machine language subroutines, but the two supplied routines let you get at nearly every hardware feature in your computer from a TINY BASIC program, including input and output directly to your peripherals.

First, how do subroutines work? In machine language a subroutine is called with a JSR instruction. This pushes the return address onto the stack and jumps to the subroutine whose address is in the JSR instruction. When the subroutine has finished its operation it executes the RTS instruction, which retrieves that return address from the stack, returning control of the computer to the program that called it. Depending on what function the subroutine is to perform, data may be passed to the subroutine by the calling program in one or more of the CPU registers, or results may be passed back from the subroutine to the main program in the same way. If the subroutine requires more data than will fit in the registers then memory is used, and the registers contain either addresses or more data. In some cases the subroutine has no need to pass data back and forth, so the contents of the registers may be ignored.

If the main program and the subroutine are both written in

TINY BASIC you simply use the GOSUB and RETURN commands to call and return from the subroutine. This is no problem. But suppose the main program is written in TINY and the subroutine is written in machine language? The GOSUB command in TINY is not implemented internally with a JSR instruction, so it cannot be used. This is rather the purpose of the USR function.

The USR function call may be written with up to three arguments. The first of these is always the address of the subroutine to be called. If you refer to USR(12345) it is the same, as if you had written a machine language instruction JSR 12345; the computer saves its return address on the stack, and jumps to the subroutine at (decimal) address 12345. For those of you who worry about such things, TINY does not actually make up a JSR with the specified address in it, but rather simulates the JSR operation with a sequence of instructions designed to have the same effect; the interpreter is clean ("pure code"), and does not modify itself.

So now we can get to the subroutine from a TINY BASIC program. Getting back is easy. The subroutine still simply executes an RTS instruction, and TINY BASIC resumes from where it left off.

If you want to pass data to the subroutine in the CPU registers, TINY allows you to do that also. This is the purpose of the second and third arguments of the USR function call. If you write a second argument in the call, this is evaluated and placed in the index register(s) of the CPU; if you write a third argument it goes into the accumulator(s). If there are results from the subroutine's operation, they may be returned in the accumulator(s) and TINY will use that as the value of the function. Thus writing the TINY BASIC statement

```
      LET P = USR (12345,0,13)
```

is approximately equivalent to writing in machine language

```
      LDX #0
      LDAA #13
      JSR 12345
      STAA P
```

Now actually there are some discrepancies. The 6800 and the 6502 are 8-bit CPUs but TINY does everything in 16-bit numbers. So in the 6502 the second argument is actually split between the X and the Y registers (the 6800 has a 16-bit index, so there is no problem), and the third argument is split between the A and B registers in the 6800 (the 6502 has no register corresponding to B, so the most significant 8 bits are discarded); the returned value is expected to be 16 bits, so the most significant 8 bits are assumed to be in the B or Y register.

It is important to realize that the three arguments in the USR function are expressions. That is, any valid combination of (decimal) numbers, variables, or function calls joined together by arithmetic operators can be used in any argument. If the variable C=6800 or C=6502 (depending on which CPU you have), the following is a perfectly valid statement in TINY BASIC.

```
13 P=P+0*USR(S+24,USR(S+20,46+C/6800),13)
```

2

When this line is executed, the inner USR call occurs first, jumping to the "PEEK" subroutine address to look at the contents of either memory location 002E or 002F (hex) (depending on whether C<6800 or not); this byte is returned as its value, and is passed immediately as the second argument of the outer call, which stores a carriage return in the memory location addressed by that byte. We are not interested in any result data from the store operation, so the result is multiplied by 0 (giving zero) and added to some variable (in this case P), which leaves that variable unchanged.

What kinds of things can we use the USR function for? As we saw in the example above, we can use it with the two built-in subroutines to "peek" or "poke" at any memory location. In particular this gives us the ability to directly access the input and output devices in the memory space.

DIRECT INPUT & OUTPUT

Suppose you have a PIA at memory address 8006-8007 hex (the B side of the PIA used by Mikbug, but any PIA will do). We want to read a 4-bit BCD digiswitch in through the low four bits, and output to a 7-segment decoded display through the high four bits. For simplicity we will read in the switch setting, add one, and output it to the display, then repeat. This program will do it:

```
      100 REM SET UP PIA DATA DIRECTION
      110 B=32768+6
      120 X=USR(S+24,B+1,0)+USR(S+24,B,240)+USR(S+24,B+1,4)
      130 REM THE FIRST USR SETS THE CONTROL REGISTER
      135 REM   TO POINT TO DATA DIRECTION REGISTER
      140 REM THE SECOND STORES HEX F0 IN IT
      150 REM THE THIRD SETS THE CONTROL REGISTER
      155 REM   TO POINT TO PERIPHERAL DATA
      160 REM X IS GARBAGE
      200 REM INPUT A NUMBER
      210 D=USR(S+20,B)
      220 REM REMOVE TRASH AND ADD ONE
      230 D=D-D/16*16+1
      240 REM OUTPUT IT
      250 X=USR(S+24,B,D*16)
      260 GOTO 200
```

You can also use the USR function for direct access to the character input and output routines, although for input you need to be careful that the characters do not come faster than your TINY BASIC program can take them. The following program inputs characters, converts lower case letters to capitals, then outputs the results:

```
      10 REM READ ONE CHARACTER
      20 A=USR(S+6)
      30 REMOVE PARITY FOR TESTING
      40 A=A-A/128*128
      50 REM IF L.C., MAKE CAPS
      60 IF A>96 IF A<123 THEN A=A-32
      70 REM OUTPUT IT
      80 A=USR(S+9,A,A)
      90 GO TO 10
```

Because of the possible timing limitations of direct character input, it may be preferable to use the buffered line input controlled by the INPUT statement of TINY. Obviously for input of numbers and expressions there is no question, but for arbitrary text input it is also useful, with a little help from the USR function. The only requirement is that the first non-blank character be a number or (capital) letter. Then the command,

```
      300 INPUT X
```

where we do not care about the value in X, will read in a line into the line buffer, affording the operator (that's you) the line editing facilities (backspace and cancel), and put what TINY thinks is the first number of the line into the variable X. Now, remembering that the line buffer is in 0030-0078 hex (approximately; the ending address varies with the length of the line), we can use the USR function and the PEEK routine (S+20) to examine individual characters at our leisure. To read the next line it is essential to convince the line scanner in TINY that it has reached the end of this line. Location 002E-002F (hex) normally contains the current pointer into the input line; if it points to a carriage return the next INPUT statement will read a new line, so all that is needed is to store a carriage return (decimal 13) in the buffer memory location pointed to by this address (see line 13 above).

STRINGS

As we have seen, character input is not such a difficult proposition with a little help from the USR function. (Character output was always easy in the PRINT statement). What about storing and manipulating strings of characters? For small strings, we can use the memory space at 0000-001F hex and 00C8-00FF hex, processing them one character at a time with the USR function. Or, if we are careful, we can fill up the beginning of the TINY BASIC program with long REM statements, and use them to hold character strings (this allows them to be initialized when the program is typed in). For example:

```
      2 REMTHIS IS A 50-CHARACTER DATA STRING FOR USE IN TINY
      3 REM0        1         2         3         4         5
      4 REM12345678901234567890123456789012345678901234567890
      5 REM...IT TAKES 56 BYTES IN MEMORY:  2 FOR THE LINE #,
      6 REM.....3 FOR THE "REM", AND ONE FOR THE TERMINAL CR.
```

If you insert one line in front to GOTO the first program line, then your program will RUN a little faster, and you do not need the letters REM at the beginning of each line (though you still need the line number and the carriage return). If you are careful, you can remove the carriage returns from all but the last text line, and the line numbers from all but the first text line (replace them with data characters), and it will look like a single line to the interpreter. Under no circumstances should you use a carriage return as a data character; if you do, none of the GOTOs, GOSUBs or RETURNs in your program will work.

Gee, you say, if it weren't for that last caveat, I could use the same technique for storing arrays of numbers.

ARRAYS

So the question arises, can the USR function help get around the fact that TINY BASIC does not have arrays? The answer is of course, yes. Obviously the small amount of space left in Page 00 and elsewhere in your system after TINY has made its memory grab is not enough to do anything useful. The possibility that one of the numbers might take on the value 13 means that you cannot use the program space. What else is there? Remember the memory bounds in 0020-0023 hex. If you start TINY with the Warm Start (S+3), you can put any memory limits you wish in here, and TINY will stay out of the rest of memory. Now you have room for array data, subroutines, or anything else. You can let the variable A hold the starting address of an array, and N the number of elements, and a bubble sort would look like this:

```
      500 LET I=1
      510 LET K=0
      520 IF USR(S+20,A+I)>=USR(S+20,A+I-1) GOTO 540
      530 K=USR(S+20,A+I)+USR(S+24,A+I,USR(S+20,A+I-1))
      535 K-USR(S+24,A+I-1,K)*0+1
      540 I=I+1
      550 IF I0 GOTO 500
      570 END
```

Of course this not the most efficient sort routine and it will be veerrry slow. But it is probably faster than writing one in machine language, even though the machine language version would execute faster.

THE STACK

A kind of sneaky place to store data is in the GOSUB stack. There are two ways to do this without messing with the Warm Start. But first let us think about the rationale.

When you execute a GOSUB, the line number of the GOSUB is saved on a stack which grows downward from the end of the user space. Each GOSUB makes the stack grow by two bytes, and each RETURN pops off the most recent saved address, to shrink the stack by two bytes. Incidentally, because the line number is saved and not the physical location in memory, you do not need to worry about making changes to your program in case of an error stop within a subroutine. Just don't remove the line that contains an unRETURNed subroutine (unless you are willing to put up with TINY's complaints).

The average program seldom needs to nest subroutines (i.e. calling subroutines from within subroutines) more than five or ten levels deep, and many computer systems are designed with a built-in limitation on the number of subroutines that may be nested. The 8008 CPU was limited to eight levels. The 6502 is limited to about

120. Many BASIC interpreters specify some maximum. I tend to feel that stack space, like most other resources, obeys Parkinson's Law: The requirements will expand to exhaust the available resources. Accordingly, the TINY BASIC subroutine nest capacity is limited only by the amount of available memory. This is an important concept. If my program is small (the program and the stack contend for the same memory space), I can execute hundreds or even thousands of

---

GOSUBs before the stack fills up. If there are no corresponding RETURN statements, all that memory just sits there doing nothing.

If you read your User's Manual carefully you will recall that memory locations 0026-0027 (hex) point to the top of the GOSUB stack. Actually they point to the next byte not yet used. The difference between that address and the end of memory (found in 0022-0023 hex) is exactly the number of bytes in the stack. One greater than the value of the top-of-stack pointer is the address of the first byte in the stack.

If you know how many bytes of data space you need, the first thing your program can do is execute half that many GOSUBs:

```
      400 REM B IS THE NUMBER OF BYTES NEEDED
      410 LET B=B-2
      420 IF B> -2 THEN GOSUB 410
      430 REM SIMPLE, ISN'T IT?
```

Be careful that you do not try to call this as a subroutine, because the return address will be buried under several hundred "420"s. If you were to add the line,

```
      440 RETURN
```

the entire stack space would be emptied before you got back to the calling GOSUB. Remember also that if you execute an END command the stack is cleared, but an error stop or a Break will not affect it. Before you start this program you should be sure the stack is clear by typing "END"; otherwise, a few times through the GOSUB loop and you will run out of memory.

If you are careful to limit it to the main program, you can grab bytes out of the stack as the need arises. An example of this is the TBIL Assembler included in this document. Whether you allocate the memory with one big grab, or a little at a time, you may use the USR peek and poke functions to get at it.

The other way to use the stack for storing data is a little more prodigal of memory, but it runs faster. It also has the advantage of avoiding the USR function, in case that still scares you. It works by effectively encoding the data in the return address line numbers themselves. The data is accessed in true stack format: last in, first out. I used this technique successfully in implementing a recursive program in TINY BASIC.

This method works best with the computed GOTO techniques described later, but the following example will illustrate the principle: Assume that the variable Q may take on the values (-1, 0, +1), and it is desired to stack Q for later use. Where this requirement occurs, use a GOTO (not a GOSUB!) to jump to the following subroutine:

```
      3000 REM SAVE Q ON STACK
      3010 IF Q<0 THEN GOTO 3100
      3020 IF Q>0 THEN GOTO 3150
      3050 REM Q=0, SAVE IT.
      3060 GOSUB 3200
      3070 REM RECOVER Q
      3080 LET Q=0
```

```
      3090 GOTO 3220
      3100 REM Q<0, SAVE IT.
      3110 GOSUB 3200
      3120 REM RECOVER Q
      3130 LET Q=-1
      3140 GOTO 3220
      3150 REM Q>0, SAVE IT.
      3160 GOSUB 3200
      3170 REM RECOVER Q
      3180 LET Q=1
      3190 GOTO 3220
      3200 REM EXIT TO (SAVE) CALLER
      3210 GOTO ...
      3220 REM EXIT TO (RECOVER) CALLER
      3230 GOTO ...
```

When the main program wishes to save Q, it jumps to the entry (line 3000), which selects one of three GOSUBs. These all converge on line 3200, which simply jumps back to the calling routine; the information in Q has been saved on the stack. To recover the saved value of Q it is necessary only to execute a RETURN. Depending on which GOSUB was previously selected, execution returns to the next line, which sets Q to the appropriate value, then jumps back to the calling routine (with a GOTO again!). Q may be resaved as many times as you like (and as you have memory for) without recovering the previous values. When you finally do execute a RETURN you get the most recently saved value of Q.

For larger numbers, the GOSUBs may be nested, each saving one bit (or digit) of the number. The following routine saves arbitrary numbers, but in the worst case requires 36 bytes of stack for each number (for numbers less than -16383):

```
      1470 REM SAVE A VALUE FROM V
      1480 IF V>=0 THEN GOTO 1490
      1482 LET V=-1-V
      1484 GOSUB 1490
      1486 LET V=-1-V
      1488 RETURN
      1490 IF V>V/2*2 THEN GOTO 1500
      1500 GOSUB 1520
      1502 LET V=V+V
      1504 RETURN
      1510 GOSUB 1520
      1512 LET V=V+V+1
      1514 RETURN
      1520 IF V=0 THEN GOTO 1550
      1522 LET V=V/2
      1524 GOTO 1490
      1550 REM GO ON TO USE V FOR OTHER THINGS
```

Note that this subroutine is designed to be placed in the path between the calling routine and some subroutine which re-uses the variable V. When the subroutine returns, it returns through the restoration part of this routine, which eventually returns to the main program with V restored. The subroutine which starts at line 1550 is assumed to be recursive, and it may call on itself through

this save routine, so that any number of instances of V may be saved on the stack. The only requirement is that to return, it first set V to 0, so that the restoration routine will function correctly. Alternatively, we could change line 1550 to jump to the subroutine start with a GOSUB:

```
1550 GOSUB ...
1552 LET V=0
1554 RETURN
```

This requires another two bytes on the stack but it removes the restriction on the exit from the recursive subroutine.

If you expect to put a hundred or more numbers on the stack in this way you might wish to consider packing them more tightly. If you use ten GOSUBs and divide by 10 instead of 2, the numbers will take one third the stack space. Divide by 41 and any number will fit in three GOSUBs, but the program gets rather long.

BIGGER NUMBERS

Sixteen bits is only good for integers 0-65535 or (-32768)-(+32767). This is fine for games and control applications, but sometimes we would like to handle fractional numbers (like dollars and cents), or very large range numbers as in scientific notation. Let's face it: Regular BASIC has spoiled us. Granted. But if you could balance your checkbook in TINY BASIC, your wife might complain less about the hundreds of dollars you spent on the computer. One common way to handle dollars and cents is to treat it as an integer number of cents. That would be OK if your balance never went over $327.67, but that seems a little unreasonable. Instead, break it up into two numbers, one for the dollars, the other for cents. Now your balance can go up to $32,767.99, which is good enough for now (if your balance goes over that you probably don't balance your own checkbook anyway). We will keep the dollars part of the balance in D and the cents in C. The following routine could be used to print your balance:

```
      900 REM PRINT DOLLARS & CENTS
      910 IF D+C<0 GOTO 960
      920 PRINT "BALANCE IS $";D;".";
      930 IF C<10 THEN PRINT 0;
      940 PRINT C
      950 RETURN
      960 PRINT "BALANCE IS -$";-D;".";
      970 IF -C<10 THEN PRINT 0;
      980 PRINT -C
      990 RETURN
```

If line number 930 is omitted, then the balance of $62.03 would print as "$62.3".

Reading in the dollars and cents is easy if you require that the operator type a comma instead of a period for a decimal point (the European tradition). If that is unacceptable, you can input the dollars part, then increment the input line buffer pointer (memory location 002E-002F hex) by one to skip over the period, then input the cents part. Be careful that it was not the carriage return you incremented over. The USR function and the peek and poke

subroutines will do all these things nicely.

Adding and subtracting two-part numbers is not very difficult. Assume that the check amount has been input to X (dollars) and Y (cents). This routine will subtract the check amount from the balance:

```
      700 REM SUBTRACT DOLLARS AND CENTS FROM BALANCE
      710 C=C-Y
      720 IF C>=0 THEN GOTO 750
      730 C=C+100
      740 D=D-1
      750 D=D-X
      760 IF D>=0 RETURN
      770 IF C=0 RETURN
      780 D=D+1
      790 C=C-100
      800 RETURN
```

Adding is a little easier because you cannot go negative (except for overflow), so it is only necessary to check for C>99; if it is, subtract 100 and add 1 to D. If your dollars and cents are in proper form (i.e. no cents values over 99), the sum will never exceed 198, so it is not necessary to retest after adjustment.

Using this same technique you can of course handle numbers with as many digits as you like, putting up to four digits in each piece. A similar technique may be used to do floating point arithmetic. The exponent part is held in one variable, say E, and the fractional part is held in one or more additional variables; in the following example we will use a four-digit fractional part in M, adding to it a number in F and N:

```
      1000 REM FLOATING POINT ADD FOR TINY BASIC
      1010 IF E-4>F THEN RETURN
      1020 IF N=0 RETURN
      1030 IF E+4F GOTO 1100
      1070 E=E+1
      1080 M=M/10
      1090 GOTO 1040
      1100 F=F+1
      1110 N=N/10
      1120 GOTO 1020
      1130 M=M+N
      1140 IF M=0 THEN E=0
      1150 IF M=0 RETURN
      1160 IF M>9999 THEN GOTO 1230
      1170 IF M>999 RETURN
      1180 IF M<-9999 THEN GOTO 1230
      1190 IF M<-999 RETURN
      1200 M=M*10
      1210 E=E-1
      1220 GOTO 1170
      1230 E=E+1
      1240 M=M/10
      1250 RETURN
```

This subroutine is a decimal floating point routine; by changing the divisors and multipliers appropriately, it can be made into a binary, hexadecimal, or even ternary floating point machine. By using the multiple precision techniques described in the checkbook balance example, greater precision can be obtained in the fractional part.

COMPUTED GOTO

One of the more powerful features of TINY BASIC is the computed line address for GOTO and G0SUB statements. A recently published[2] set of games to run in TINY had several large blocks of the program devoted to sequences of IF statements of the form:

```
110 IF I=1 GOTO 1000
120 IF I=2 GOTO 2000
130 IF I=3 GOTO 3000
140 IF I=4 GOTO 4000
150 GOTO 100
```

Now there is nothing wrong with this form of program, but I'm too lazy to type all that, and besides, I could not get the whole program into my memory. Instead of lines 110-140 above, the single line

```
125 IF I>0 IF I<5 GOTO I*1000
```

does exactly the same thing in less memory, and probably faster.

Another part of this program simulated a card game, in which the internal numbers 11-14 were recognized (using the same kind of sequence of IFs) in three different places, and for each different number the name of the corresponding face card was printed. The astonishing thing was that the sequence of IFs, PRINTs, and GOTOs was repeated three different places in the program. Now I'm glad that Carl enjoys using TINY BASIC, and that he likes to type in large programs to fill his voluminous memory; but as I said I'm lazy, and I would rather type in one set of subroutines:

```
10110 PRINT "JACK"
10115 RETURN
10120 PRINT "QUEEN"
10125 RETURN
10130 PRINT "KING"
10135 RETURN
10140 PRINT "ACE"
10145 RETURN
```

Then in each of the three places where this is to be printed, use the simple formula,

```
2510 GOSUB 10000+B*10
```

Along the same line, when memory gets tight you may be able to save a few bytes with a similar technique. Suppose your program has thirteen "GO TO 1234" statements in it; if you have an unused variable (say, U) you can, in the direct execution mode, assign it the value 1234 (i.e. the line number that all those GOTOs go to).

Then replace each "GO TO 1234" with a "GOTOU" squeezing out the extra spaces (TINY BASIC ignores them anyway). This will save some thirty or forty bytes, and it will probably run faster also.

EXECUTION SPEED

TINY BASIC is actually quite slow in running programs. That is one of the hazards of a two-level interpreter approach to a language processor. But there are some ways to affect the execution speed. One of these is to use the keyword "LET" in your assignment statements. TINY BASIC will accept either of the following two forms of the assignment statement and do the same thing,

```
      R=2+3
      LET R=2+3
```

but the second form will execute much faster because it is unnecessary for the interpreter to first ascertain that it is not a REM, RUN, or RETURN statement. In fact, the LET keyword is the first tested, so that it becomes the fastest-executing statement, whereas the other form must be tested against all twelve keywords before it is assumed to be an assignment statement.

Another way to speed up program execution depends on the fact that constant numbers are converted to binary each time they are used, while variables are fetched and used directly with no conversion. If you use the same constant over and over and you do not otherwise use all the variables, assigning that number to one of the spare variables will make the program both shorter and faster. You can even make the assignment in an unnumbered line; the variables keep their values until explicitly changed.

Finally it should be noted that GOTOs and GOSUBs always search the program from the beginning for their respective line numbers. Put the speed-sensitive part of the program near the front, and the infrequently used routines (setup, error messages, and the like) at the end. This way the GOTOs have fewer line numbers to wade through so they will run faster.

DEBUGGING

Very few programs run perfectly the first time. When your program does not seem to run right there are several steps you can take to find the problem.

First of all, try to break it up into its component parts. Use the GOTO command and the END statement to test each part separately if you can. Add extra PRINT statements along the way to print out the variables you are using; sometimes the variables do not have the values in them that we expected. Also, the PRINT statements will give you an idea as to the flow of execution. For example, in testing the sort program above (lines 500-570) I inserted the following extra PRINT statements:

```
      525 PR "X";
      545 PR ".";
      555 PR
```

This gave me an idea where in the sort algorithm I was, so I could follow the exchanges (the "X"s), where each line represented one pass through the main loop. Endless loops become more obvious this way.

If you have not used all the sequential line numbers, you can insert breakpoints in the program in the form of a line number with an illegal statement -- I like to use a single period, because it is easy to type and does not take much space:

```
      10 LET A=B+1234
      11 .
      20 GOSUB 100+A
```

Here when you type RUN, the program will stop with the error message,

```
      !184 AT 11
```

Now we can PRINT A, B, etc. to see what might be wrong, or type in GOTO 20 to resume, with no loss to the original program.

As we have seen, there is not much that TINY BASIC cannot do (except maybe go fast). Sure, it is somewhat of a nuisance to write all that extra code to get bigger numbers or strings or arrays, but you can always code up subroutines which can be used in several different programs (like the floating point add above (lines 1000-1250), then save them, off on paper tape or cassette.

Remember, your computer (with TINY BASIC in it) is limited only by your imagination.

REFERENCES

[1] TINY BASIC User's Manual. Available from ITTY BITTY COMPUTERS, P.O. Box 23189, San Jose, CA 95153.

[2] Doctor Dobb's Journal, v1 No.7, p,26. Available from PCC, P.O. Box 310, Menlo Park, CA 94025.

TBIL -- The TINY BASIC Interpreter Language

The TINY BASIC interpreter is, in the words of Dennis Allison who conceived it, something like an onion. There is an inner machine language program (ML) which interprets a second program written in an intermediate language (IL), which in turn interprets the BASIC program, and so on. This document describes that intermediate language and the virtual machine which executes it.

The IL interpreter is a pure interpreter in the sense that the entire BASIC interpreter is implemented within the bounds of the language. There are no deus ex machina escapes to machine language other than the well-defined machine-language subroutine call. The language is substantially the same as that defined by Dennis Allison in Dr. Dobb's Journal and PCC.

Most of the instructions in the IL occupy one byte of code. A few instructions may be followed by one or more bytes of immediate data and there are two jump instructions which are actually two bytes in length.

The interpreter itself uses no variable storage. Computations are performed on an expression stack, so that all procedures are capable of recursion. The interpreter does have access to memory Page 00 for data storage, but little use is made of this capability.

The IL code is self-relative. That is, all jumps are relative to the beginning of the IL interpreter. Thus the code can be moved to another part of memory without re-assembling it. The conditional branches are PC-relative and branch only forward to a maximum displacement of 31 bytes. An unconditional branch has a range of 31 bytes forward or backwards. Since the interpreter is generally quite small this is not a serious limitation. There are only two or three places where a longer conditional branch would be needed; these are accommodated by branching to a jump. The jumps have an address space of 11 bits, or 2K bytes from the beginning of the IL code.

Two of the instructions include a literal text string as part of the code. This string follows the opcode and is of arbitrary length. The end of the string is signaled by the eighth bit on the last byte being set to one. Since the text is generally assumed to be ASCII, a 7-bit code, this is a reasonable way to save space.

There are two stacks in the virtual machine. The computational or Expression Stack has already been mentioned. The other stack is a control stack, used to hold subroutine return addresses. The same control stack is used for subroutine returns in three languages: BASIC, IL, and ML. Thus a certain amount of care is necessary in the maintenance of this stack. The ML interpreter will take care of its

requirements by placing them on the stack top, and a special parameter ("SPARE") in the main program defines the maximum amount of space to be left on the stack for this purpose. Overflow of this part of the stack is not detectable, so it is essential that the reserved space be sufficiently large. Because the ML interpreter is not recursive this is reasonably safe, provided that the stack requirements of the I/O are known and limited. Beneath the ML stack is the IL stack. Subroutine calls in the IL have their return address pushed onto this stack. The ML interpreter does check for stack overflow by measuring the distance between the top of the IL stack and the end of the BASIC program; if this ever becomes less than the SPARE parameter, the stack is considered to have overflowed.

Beneath the IL stack is the BASIC stack. This holds the line numbers for GOSUB lines as they are executed. There are two instructions in the IL for accessing the top element of this stack (i.e. a Push and a Pop). For these instructions to work properly it is essential that the IL stack be empty. No check is made in the ML for this condition, and it is the responsibility of the IL program to insure this form of stack integrity. In other words, BASIC language GOSUBs and RETURNs cannot be processed within an IL subroutine.

The interpretation of the BASIC program is tied to a pointer (invisible to the IL) which points to the current character in the current line. The IL has no direct control over this pointer, but several of the IL instructions cause it to be advanced or otherwise modified. In particular it may be changed to point to the beginning of another BASIC line to implement the BASIC sequence control operations (GOTO, GOSUB, RETURN). It may also be exchanged with its logical dual, a pointer to the current character in the input line buffer. This permits the same interpreter to operate on the BASIC program stored in memory or on a direct execution statement in the line buffer.

There are a number of IL operations which may result in an error condition. All errors abort the IL execution and print out on the console the IL program counter at which the error occurred. If the program execution flag was set, the most recently accessed BASIC line number is also typed out in the message. The relative address (relative to the beginning of the IL) becomes the error number in the error message. Since only one error is possible in most operations, this gives a unique identification of the difficulty. The IL address is printed in decimal and represents the address of the next byte which would have been executed but for the error. There is one operation which has two possible failure modes; for one of these the IL address is decremented before printing to distinguish it from the other. Error stops may be explicitly requested in the IL program by the execution of a branch with a zero offset.

After typing out the error addresses, the ML interpreter clears the ML and IL stacks (but not the BASIC stack!) and restarts the IL interpreter at relative address 0. Nothing else is changed, except that the execution flag is cleared, putting the interpreter into the command mode. When the Break condition is recognized in advancing to the next BASIC statement this is treated by the ML as

an error condition, after forcing the IL program counter to relative zero.

The ML interpreter maintains a flag to distinguish program RUN mode from direct statement execution (command mode). Advancing to the next statement (an IL instruction) examines this flag, and if in the command mode, the IL program is restarted at the beginning. If the flag is set in the RUN mode, execution resumes at the IL address saved by the Execute instruction. It is important that the Execute instruction be given in the IL before any Next BASIC statement advance, but once it has been done there is no restriction (i.e. the saved address is never lost).

The Break condition is tested only during the execution of the statement advance (Next), so that resumption of an interrupted program leaves no computational gaps. The Break condition is also tested during a LIST operation, but only to abort the listing; if the LIST occurred within program execution (i.e. with the RUN mode flag set), a second Break condition is required to terminate the program.

The following is a detailed description of the operation of each of the IL opcodes. With each description is also given the hexadecimal opcode and the mnemonic recognized by the assembler. Not all of the opcodes are defined. Some have been incorporated into unused functions; others are reserved for possible future expansion and execute as NOPs.

INTERPRETIVE LANGUAGE OPERATION CODES

```
SX n    00-07   Stack Exchange.

                Exchange the top byte of computational stack with
that "n" bytes into the stack. The top byte of the stack is
considered to be byte 0, so SX 0 does nothing. The sequence of
instructions
                          SX 1
                          SX 3
                          SX 1
                          SX 2

may be used to exchange the top two numbers (two bytes each) on the
stack. Only the top eight bytes on the stack are accessible, to this
instruction. If the stack is empty an error stop may or may not
occur, depending on which ML interpreter is implemented.

NO      08      No Operation.
                This may be used as a space filler (such as to
ignore a skip).

LB n    09nn    Push Literal Byte onto Stack.
                This adds one byte to the computational stack, which
is the second byte of the instruction. An error stop will occur if
the stack overflows.

LN n    0Annnn  Push Literal Number.
                This adds the following two bytes to the
computational stack, as a 16-bit number. Stack overflow results in
an error stop.

DS      0B      Duplicate Top Number (two bytes) on Stack.
                An error stop will occur if there are less than two
bytes on the expression stack or if the stack overflows.

SP      0C      Stack Pop.
                The top two bytes are removed from the computational
stack and discarded. Underflow results in an error stop.

SB      10      Save BASIC Pointer.
                If the BASIC pointer is pointing into the input line
buffer, it is copied to the Saved Pointer; otherwise the two
pointers are exchanged.

RB      11      Restore BASIC Pointer.
                If the Saved Pointer is pointing into the input line
buffer, it is replaced by the value in the BASIC pointer; otherwise
the two pointers are exchanged.
          Normally the Saved Pointer will point to the next item in the
input line buffer while the BASIC pointer points to the program
being executed. When an INPUT instruction in BASIC is interpreted,
the two pointers are exchanged by the SB opcode so that the
expression handling capabilities of the interpreter may be applied
to the input data, then the pointers are restored (exchanged again)
```

```
by the RB. In direct execution (command mode) the BASIC pointer is
```

```
already in the input line buffer, and the contents of the Saved
Pointer are meaningless; in this case the SB instruction does not
alter the BASIC pointer, and the RB opcode should leave both
pointers pointing to the next item in the input string.

FV      12      Fetch Variable.
                The top byte of the computational stack is used to
index into Page 00. It is replaced by the two bytes fetched. Error
stops occur with stack overflow or underflow.

SV      13      Store Variable.
                The top two bytes of the computational stack are
stored into memory at the Page 00 address specified by the third
byte on the stack. All three bytes are deleted from the stack.
Underflow results in an error stop.

GS      14      GOSUB Save.
                The line number on the current BASIC line is pushed
onto the BASIC region of the control stack. It is essential that
the IL stack be empty for this to work properly but no check is
made for that condition. An error stop occurs on stack overflow.

RS      15      Restore Saved Line.
                Pop the top two bytes off the BASIC region of the
control stack, making them the current line number. Set the BASIC
pointer at the beginning of that line. Note that this is the line
containing the GOSUB which caused the line number to be saved. As
with the GS opcode, it is essential that the IL region of the
control stack be empty. If the line number popped off the stack does
not correspond to a line in the BASIC program an error stop occurs.
An error stop also results from stack underflow.

GO      16      GOTO.
                Make current the BASIC line whose line number is
equal to the value of the top two bytes in the expression stack.
That is, the top two bytes are popped off the computational stack,
and the BASIC program is searched until a matching line number is
found. The BASIC pointer is then positioned at the beginning of that
line and the RUN mode flag is turned on. Stack underflow and
non-existent BASIC line result in error stops.

NE      17      Negate (two's complement).
                The number in the top two bytes of the expression
stack is replaced with its negative.

AD      18      Add.
```

```
                Add the two numbers represented by the top four
bytes of the expression stack, and replace them with the two-byte
sum. Stack underflow results in an error stop.

SU      19      Subtract.
                Subtract the two-byte number on the top of the
expression stack from the next two bytes and replace the four bytes
with the two-byte difference. This is exactly equivalent to the
two-instruction sequence
                          NE
                          AD
```

```
and has the same error stop on underflow.

MP      1A      Multiply.
                Multiply the two numbers represented by the top four
bytes of the computational stack, and replace them with the least
significant 16 bits of the product. Stack underflow is possible.

DV      1B      Divide.
                Divide the number represented by the top two bytes
of the computational stack into that represented by the next two.
Replace the four bytes with the quotient and discard the remainder.
This is a signed (two's complement) integer divide, resulting in a
signed integer quotient. Stack underflow or attempted division by
zero result in an error stop.

CP      1C      Compare.
                The number in the top two bytes of the expression
stack is compared to (subtracted from) the number in the fourth and
fifth bytes of the stack, and the result is determined to be
Greater, Equal, or Less. The low three bits of the third byte mask a
conditional skip in the IL program to test these conditions; if the
result corresponds to a one bit, the next byte of the IL code is
skipped and not executed. The three bits correspond to the
conditions as follows:
        bit 0   Result is Less
        bit 1   Result is Equal
        bit 2   Result is Greater
Whether the skip is taken or not, all five bytes are deleted from
the stack. This is a signed (two's complement) comparison so that
any positive number is greater than any negative number. Multiple
conditions, such as greater-than-or-equal or unequal (i.e. greater-
than-or-less-than), may be tested by forming the condition mask byte
of the sum of the respective bits. In particular, a mask byte of 7
will force an unconditional skip and a mask byte of 0 will force no
skip. The other five bits of the control byte are ignored. Stack
underflow results in an error stop.
```

```
NX      1D      Next BASIC Statement.
                Advance to the next line in the BASIC program, if in
the RUN mode, or restart the IL program if in the command mode. The
remainder of the current line is ignored. In the Run mode if there
is another line it becomes current with the pointer positioned at
its beginning. At this time, if the Break condition returns true,
execution is aborted and the IL program is restarted after printing
an error message. Otherwise IL execution proceeds from the saved IL
address (see the XQ instruction). If there are no more BASIC
statements in the program an error stop occurs.

LS      1F      List The Program.
                The expression stack is assumed to have two 2-byte
numbers. The top number is the line number of the last line to be
listed, and the next is the line number of the first line to be
listed. If the specified line numbers do not exist in the program,
the next available line (i.e. with the next higher line number) is
assumed instead in each case. If the last line to be listed comes
```

```
before the first, no lines are listed. If the Break condition comes
true during a List operation, the remainder of the listing is
aborted. Zero is not a valid line number, and an error stop occurs
if either line number specification is zero. The line number
specifications are deleted from the stack.

PN      20      Print Number.
                The number represented by the top two bytes of the
expression stack is printed in decimal with leading zero
suppression. If it is negative, it is preceded by a minus sign
(hyphen) and the magnitude is printed. Stack underflow is possible.

PQ      21      Print BASIC String.
                The ASCII characters beginning with the current
position of the BASIC pointer are printed on the console. The string
to be printed is terminated by the quotation mark ("), and the BASIC
pointer is left at the character following the terminal quote. An
error stop occurs if a carriage return is imbedded in the string.

PT      22      Print Tab.
                Print one or more spaces on the console, ending at
the next multiple of eight character positions (from the left
margin).

NL      23      New Line.
                Output a carriage-return-linefeed sequence to the
console.
```

```
PC "xxxx"  24xxxxxxXx   Print Literal String.
                        The ASCII string follows the opcode and its
last byte has the most significant bit set to one. The character
string is output to the console unmodified; that is, all eight bits
of each byte is output, so that the last byte and only that byte is
output with the parity bit set to one. This of course may be altered
by the output routine.

GL      27      Get Input Line.
                ASCII characters are accepted from the console input
to fill the line buffer. If the line length exceeds the available
space, the excess characters are ignored and bell characters are
output. The line is terminated by a carriage return. NUL and DEL
codes (hex 00 and FF) are ignored; linefeed and DC3 respectively
turn the "tape mode" on and off. Any characters which match the
Backspace parameter result in the deletion of the previous character
in the line buffer, if any; if the line buffer is empty the effect
is that of a cancel. Any character which matches the Cancel
parameter stores a carriage return in the first position of the line
buffer and terminates the input. On completing one line of input,
the BASIC pointer is set to point to the first character in the
input line buffer, and a carriage-return-linefeed sequence is
output.

IL      2A      Insert BASIC Line.
                Beginning with the current position of the BASIC
pointer and continuing to the carriage return, the line is inserted
into the BASIC program space; for a line number, the top two bytes
```

```
of the expression stack are used. If this number matches a line
already in the program it is deleted and the new one replaces it. If
the new line consists of only a carriage return, it is not inserted,
though any previous line with the same number will have been
deleted. The lines are maintained in the program space sorted by
line number. If the new line to be inserted is a different size than
the old line being replaced, the remainder of the program is shifted
over to make room for it or to close up the gap as necessary. If
there is insufficient memory to fit in the new line, the program
space is unchanged and an error stop occurs (with the IL address
decremented). A normal error stop occurs on expression stack
underflow or if the number is zero, which is not a valid line
number. After completing the insertion, the IL program is restarted
in the command mode.

MT      2B      Mark the BASIC program space Empty.
                Also clears the BASIC region of the control stack
and restart the IL program in the command mode. The memory bounds
and stack pointers are reset by this instruction to signify an empty
```

```
program space, and the line number of the first line is set to zero,
which is the indication of the end of the program. The remainder of
the program is not altered, though it is now vulnerable to intrusion
by the control stack. The program may be recovered if accidentally
CLEARed by storing a non-zero line number in the first two bytes of
the BASIC program space, then requesting a LIST. If this is made on
a machine-readable medium, it may be reloaded. Any execution of the
IL instruction after a MT instruction will destroy the contents of
memory not enclosed by the program bounds in locations 0020-0025.

XQ      2C      Execute.
                Turns on RUN mode. This instruction also saves the
current value of the IL program counter for use of the NX
instruction, and sets the BASIC pointer to the beginning of the
BASIC program space. An error stop occurs if there is no BASIC
program. This instruction must be executed at least once before the
first execution of a NX instruction.

WS      2D      Stop.
                Stop execution and restart the IL program in the
command mode. The entire control stack (including the BASIC region)
is also vacated by this instruction. This instruction effectively
jumps to the Warm Start entry of the ML interpreter.

US      2E      Machine Language Subroutine Call.
                The top six bytes of the expression stack contain
three numbers with the following interpretations: The top number is
loaded into the A (or A and B) register; the next number is loaded
into 16 bits of Index register; the third number is interpreted as
the address of a machine language subroutine to be called using the
normal subroutine call sequence (which is simulated for this purpose
by the ML interpreter). These six bytes on the expression stack are
replaced with the 16-bit result returned by the subroutine. Stack
underflow results in an error stop.

RT      2F      IL Subroutine Return.
                The IL control stack is popped to give the address
```

```
of the next IL instruction. An error stop occurs if the entire
control stack (IL and BASIC) is empty.

JS a    3000-37FF       IL Subroutine Call.
                        The least significant eleven bits of this
2-byte instruction are added to the base address of the IL program
to become the address of the next instruction. The previous contents
of the IL program counter are pushed onto the IL region of the
control stack. Stack overflow results in an error stop.
```

```
J a     3800-3FFF       Jump.
                        The low eleven bits of this 2-byte
instruction are added to the IL program base address to determine
the address of the next IL instruction. The previous contents of the
IL program counter is lost.

BR a    40-7F   Relative Branch.
                The low six bits of this instruction opcode are
added algebraically to the current value of the IL program counter
to give the address of the next IL instruction. Bit 5 of the opcode
is the sign, with + signified by 1, - by 0. The range of this branch
is 31 bytes from address of the byte following the opcode, in either
direction. An offset of zero (i.e. opcode 60) results in an error
stop. The branch operation is unconditional.

BC a "xxx"   80xxxxXx-9FxxxxXx  String Match Branch.
                                The ASCII character string in the IL
following this opcode is compared to the string beginning with the
current position of the BASIC pointer, ignoring blanks in the BASIC
program. The comparison continues until either a mismatch is found,
or an IL byte is reached with the most significant bit set to one.
This is the last byte of the string in the IL, and it is compared as
a 7-bit character; if equal, the BASIC pointer is positioned after
the last matching character in the BASIC program and the IL program
continues with the next instruction in sequence. Otherwise the BASIC
pointer is not altered and the low five bits of the Branch opcode
are added to the IL program counter to form the address of the next
IL instruction. If the strings do not match and the branch offset is
zero an error stop occurs.

BV a    A0-BF   Branch if Not Variable.
                If the next non-blank character pointed to by the
BASIC pointer is a capital letter, its ASCII code is doubled and
pushed onto the expression stack and the IL program advances to the
next instruction in sequence, leaving the BASIC pointer positioned
after the letter; if not a letter the branch is taken and the BASIC
pointer is left pointing to that character. An error stop occurs if
the next character is not a letter and the offset of the branch is
zero, or on stack overflow.

BN a    C0-DF   Branch if Not a Number.
                If the next non-blank character pointed to by the
BASIC pointer is not a decimal digit, the low five bits of the
opcode are added to the IL program counter, or if zero an error stop
occurs. If the next character is a digit, then it and all decimal
digits following it (ignoring blanks) are converted to a 16-bit
```

```
binary number which is pushed onto the expression stack. In either
```

```
case the BASIC pointer is positioned at the next character which is
neither blank nor digit. Stack overflow will result in an error
stop.

BE a    E0-FF   Branch if Not Endline.
                If the next non-blank character pointed to by the
BASIC pointer is a carriage return, the IL program advances to the
next instruction in sequence; otherwise the low five bits of the
opcode (if not zero) are added to the IL program counter to form the
address of the next IL instruction. In either case the BASIC pointer
is left pointing to the first non-blank character encountered; this
instruction will not pass over the carriage return, which must
remain for testing by the NX instruction. As with the other
conditional branches, the branch may only advance the IL program
counter from 1 to 31 bytes; an offset of zero results in an error
stop.
```

TBIL ASSEMBLER

To aid in developing and modifying the IL program an assembler has been written in TINY BASIC. This assembler accepts the mnemonics for the IL assembly language and outputs a hexadecimal object code suitable for loading into memory. It is a two-pass assembler, building the symbol table on the first pass and generating the full hex object code on the second pass.

Since TINY BASIC does not allow strings or arrays, the source file and the symbol table are manipulated using the USR function to call on the standard machine language subroutines to load and store bytes in memory. This is unfortunately very slow, so a third subroutine, which loads two bytes, is also used in an effort to speed things up a little. Comments in the source listing of the assembler indicate how such a routine may be coded. The assembler is still compute-bound, and can be expected to take several hours on each pass. This is considered acceptable only because of the infrequent need to assemble the IL code.

The assembler accepts free-form input with two kinds of source lines: Comment lines and program instruction lines. Each line of either kind must begin with a line number. This is actually a kludge to convince TINY BASIC to read the source line with an INPUT command, and the number has no significance to the assembler other than that it is zero on the last line of the program.

Comment lines are indicated to the assembler by a period following the line number. They are not processed further.

Instruction lines may begin with a label or not. A label is signified by a leading colon (which is not part of the label) followed by a letter and up to three more letters and/or digits, and terminated by a blank.

The next field after the label, or the first field of a line without a label, is the instruction mnemonic. This is one of the two-letter codes (or one letter in the case of J) defined earlier.

The instructions which require operands should be followed by at least one blank, then the operand in the correct format. Jumps and branches accept a label reference; the branches also accept the single symbol "*" to signify an error stop branch. The SX instruction requires a single octal digit (1-7).

The LB and LN instructions should be followed by a decimal number. This number is processed by the BASIC INPUT command which accepts expressions and ignores blanks, so care must be taken in what is allowed to follow the number. In particular it may not be followed by more decimal digits or the characters + - * or /. The number must start with a digit.

The BC and PC instructions are followed by a string (after the label in the case of BC). The string is enclosed in a pair of delimiters which may be any non-blank character except the ASCII circumflex (hex 5E which sometimes prints as an up-arrow). Any character within the string which is followed by a circumflex has a hex 40 subtracted from its code, making it possible to generate strings with control characters in them. The last character of the string has the most significant bit set to one in the object code.

Everything on the source line after the operands, if any, is treated by the assembler as comments.

The operation of the assembler is shaped by the restrictions imposed by TINY BASIC. The source lines must not be larger than 60 or so characters to leave room in the expression stack. Each source line must end in a DC3 control (X-OFF) unless other reader control is used, since several tens of seconds are required to process each line.

The program is loaded and started with a RUN. It will ask for the addresses of the byte load and store routines, which should be typed in in decimal. It will also ask for the memory address that the program is to load into. This address is only used in the generation of the location counter output and has no effect on the code generation.

One of the first things done in the assembler is to search for the mnemonic table, which is imbedded in pseudo-comment lines near the beginning of the assembler. These are identified by the leading asterisk on the line, although the search is keyed to line number 3. The symbol table is also initialized at empty.

Each line of the assembled program will have the hexadecimal memory address, the hexadecimal object code to be loaded into that address, a semicolon marking the end of the machine code, then the next source line. Notice that the source line is echoed as it is read (this is done by the I/O routines), so the assembled code for that line is at the beginning of the next line. If the source file contains a linefeed character after each carriage return, then the object code will appear on the same line in the listing, but in fact the object code follows it in the output file. In the case of the LN, PC, and BC opcodes, which generate more than two bytes of code, a second line will be used for the excess object code. The listing produced for Pass 1 will look very much like that for Pass 2, except that some of the object code will be incomplete.

Assembly errors which do not crash the program will be identified by a two letter indication enclosed in a pair of asterisks. The following is a summary of the errors recognized and flagged by this assembler:

```
*DL* Duplicate label (Pass 1 only)
```

```
        *IE* Unidentifiable mnemonic
        *OP* Incorrectly formed Operand
        *US* Undefined symbol in jump or branch
        *LE* Premature line end
```

Some source program errors will be trapped by the TINY BASIC interpreter and halt the assembler. These are catastrophic in the sense that not only is the assembly aborted, but the remainder of the source file is loaded by TINY into memory over the assembler as if it were a BASIC program, thus destroying the integrity of the assembler. Errors which are catastrophic are:

```
        Lines without a line number
        Excessively long lines
        Invalid expression as the operand of LN or LB
        Symbol table overflow
```

This version of the assembler may be expected to run in something under 8K bytes of memory, depending on how many of the comment lines and excess blanks are removed.

Operationally, the program is fairly direct with few tricky kludges.
The symbol table is built by the assembler by stealing space out of the GOSUB stack. For each label to be added to the table, three unRETURNed GOSUBs are executed, making six bytes available. Symbols with less than 4 characters are filled out with spaces. The same symbol table search routine is used for both definition (to check for duplicates) and reference. The table is searched with the memory fetch USR commands.
The opcode table is searched in a similar way. The hex codes are never actually converted to binary, but a special subroutine selects the appropriate digit printing statement based on the ASCII value of the codes. In the few cases where the operand is imbedded into the opcode, the extra bits are added in before output.
The type of instruction (i.e. the kind of operands accepted for the particular instruction) is determined by its position in the table: The first position is SX; the next two are jumps; the next five are branches, followed by the string opcodes (note the overlap). The literal byte and number opcodes are finally followed by all the generics (no operand). The assembler knows how many opcodes there are, and stops looking when this count is reached, rather than looking for some end-of-table flag. The table is broken up into several lines of TINY BASIC; the line boundaries are aligned with the mnemonic positions in the table, so they represent opcodes which never match (the mnemonic would be CR-NUL).

The operation of the remainder of the assembler is fairly self-evident and needs no further discussion.

```
1 REM TINY BASIC IL ASSEMBLER VERSION 0        1 JAN 1977
2 GOTO 100
3 *SX00JS30J 38BR40BVA0BNC0BEE0BC80PC24LB09LN0ANO08
4 *DS0BSP0CSB10RB11FV12SV13GS14RS15GO16NE17AD18SU19MP1ADV1B
5 *CP1CNX1DLS1FPN20PQ21PT22NL23GL27IL2AMT2BXQ2CWS2DUS2ERT2F
6
7              ....COPYRIGHT (C) 1977  BY TOM PITTMAN....
10 REMARKS:
11 LINES 3-5 ARE OPCODE TABLE
12 LABEL TABLE USES GOSUB STACK
13 .
14 THIS PROGRAM USES A 2-BYTE PEEK USR FUNCTION
15 PUT ITS ADDRESS IN VARIABLE D.
16 IN 6800:
17   LDA A,1,X        A IS LSB
18   LDA B,0,X
19   RTS
20 IN 6502:
21   STX $C3          ($C2=00)
22   LDA ($C2),Y      GET MSB
23   PHA              SAVE IT
24   INY
```

```
25   LDA ($C2),Y      GET LSB
26   TAX
27   PLA
28   TAY              Y=MSB
29   TXA
30   RTS
31 NOTE THAT THIS PROGRAM CORRECTS FOR 2-BYTE-DATA
32 IN 6502 FORMAT (LSB,MSB) WHEN INITIALIZING.
33 .
34 THE FOLLOWING VARIABLES ARE DEFINED:
35 A  STARTING ADDRESS
36 B  LINE BUFFER POINTER ADDRESS
37 C  LINE POINTER WORK
38 D  2-BYTE PEEK USR FUNCTION ADDRESS
39 E  END OF OPCODE TABLE
40 F  PASS #
41 G  PEEK USR FUNCTION ADDRESS
42 H  HEX WORK
43 I  TEMP WORK
44 J  TEMP WORK
45 K  TEMP WORK (HEX)
46 L  (RELATIVE) LOCATION COUNTER
47 M
48 N  LINE NUMBER
49 O  OP TABLE START
50 P  POKE USR FUNCTION ADDRESS
51 Q
52 R
53 S  SYMBOL TABLE START
54 T  TEMP (TABLE POINTER)
55 U
56 V  SYMBOL WORK
57 W  SYMBOL WORK
```

```
58 X ERROR COUNT
59 Y
60 Z
61 .
62 SOURCE FILE IS IN THE FORM
63 (LINE NUMBER) :LABEL OP OPND COMMENTS
64 THE LINE NUMBER MUST BE >0.
65 THE LABEL IS IDENTIFIED BY THE LEADING COLON,
66 AND MAY BE 1-4 CHARACTERS LONG (FIRST IS LETTER);
67 IT IS TERMINATED BY BLANK, AND MAY BE OMITTED.
68 .
69 OP IS THE 2-LETTER OPCODE.
70 OPND IS THE OPERAND:
71 FOR SX IT MUST BE A DIGIT 1-7
```

```
72 FOR LB OR LN, A DECIMAL NUMBER 0-255 OR 0-65535
73 FOR PC, A STRING OF THE FORM 'STRING'
74 FOR JUMPS & BRANCHES IT MUST BE A SYMBOL
75 BRANCHES MAY REFER TO SYMBOL "*"
76 TO INVOKE ERROR STOP FORM.
77 BC REQUIRES BOTH A SYMBOL AND A STRING,
78 SEPARATED BY ONE OR MORE SPACES.
79 COMMENTS SHOULD BE PRECEDED BY A SPACE,
80 AND SHOULD NOT BEGIN WITH A DIGIT OR (+,-,*,/)
81 COMMENT LINES HAVE A PERIOD
82 FOLLOWING THE LINE NUMBER.
83 THE END OF FILE IS A LINE NUMBER 0.
84 .
85 SOURCE IS LISTED ON BOTH PASSES.
86 OUTPUT IS: HEX ADDRESS, HEX CODE, SEMICOLON,
87 ON SAME LINE AS FOLLOWING SOURCE.
88 .
89 .
90 ERROR FLAGS:
91 *DL* DUPLICATE LABEL (PASS 1)
92 *OP* OPERAND FORMAT ERROR
93 *IE* UNDEFINED OP CODE
94 *LE* INCOMPLETE LINE
95 *US* UNDEFINED SYMBOL (PASS 2)
99 .
100 REM
101 REM LINES 101-199 ONLY NEED TO EXECUTE ONCE.
102 REM THEY SHOULD BE DELETED AT STOP.
103 REM INPUT ADDRESS CONSTANTS
104 PRINT "PLEASE TYPE IN USR ADDRESS FOR PEEK (IN DECIMAL)";
105 INPUT G
106 PRINT "ADDRESS FOR POKE";
107 INPUT P
108 PRINT "ADDRESS FOR 2-BYTE PEEK";
109 INPUT D
110 B=47
111 O=USR(D,32)
112 E=USR(D,34)
113 IF USR(G,B)>0 GOTO 118
114 B=46
115 O=USR(G,32)+USR(G,33)*256
116 E=USR(G,34)+USR(G,35)*256
```

```
118 E=E+1
119 REM FIND OPCODE TABLE (LINE 3)
120 O=O+1
121 IF USR(G,O)<>3 GOTO 120
122 O=O+2
```

```
130 Y=1
131 N=0
132 PRINT "DO YOU NEED INSTRUCTIONS (Y OR N)";
133 INPUT I
134 IF I=Y, LIST 61,99
190 PRINT "REMOVE LINES 10-99, 101-199"
191 PRINT "OR IF YOU HAVE PLENTY OF MEMORY,"
192 PRINT "RETYPE LINE: 100 GOTO 200"
193 PRINT "THEN TYPE RUN."
198 END
199 REM 2-PASS ASSEMBLER. START FIRST PASS.
200 X=0
201 S=E
202 F=0
203 PRINT "(DECIMAL) STARTING ADDRESS";
204 INPUT A
205 F=F+1
206 IF F=3 GOTO 760
207 L=0
208 PRINT
209 PRINT "TBIL ASSEMBLER, PASS ";F
210 PRINT
211 GOSUB 460
212 PRINT ";    ";
213 REM GET NEXT INPUT LINE
214 I=USR(P,USR(G,B),13)
215 INPUT N
216 REM LINE NUMBER 0 IS EOF
217 IF N=0 GOTO 205
218 GOSUB 460
219 REM CHECK FOR COMMENT
220 I=USR(G,USR(G,B))
221 IF I<58 GOTO 212
222 REM PROCESS LABEL, IF ANY
223 IF I>64 GOTO 300
224 GOSUB 405
225 GOSUB 500
231 REM CHECK FOR DUPLICATES ON PASS 1
232 IF F>1 GOTO 300
234 IF T=0 GOSUB 237
235 GOTO 901
237 GOSUB 238
238 GOSUB 239
239 S=S-6
240 REM INSERT THIS ONE
241 I=USR(P,S,V/256)+USR(P,S+1,V)
242 I=USR(P,S+2,W/256)+USR(P,S+3,W)
243 I=USR(P,S+4,L/256)+USR(P,S+5,L)
290 REM LOOK AT OPCODE
300 GOSUB 410
301 IF I<65 GOTO 911
```

```
305 I=USR(D,USR(G,B))
306 GOSUB 404
307 REM SEARCH OPCODE TABLE
308 T=O
309 IF USR(D,T)=I GOTO 313
310 T=T+4
311 IF T48 IF I<56 GOTO 323
333 REM OPERAND FORMAT ERROR
334 GOTO 921
336 IF F=1 GOTO 212
337 GOTO 931
339 REM JUMP & CALL
340 L=L+1
341 GOSUB 410
342 IF I<65 GOTO 334
344 K=W-W/16*16
345 GOSUB 500
346 IF T=0 GOTO 336
347 K=I+(K+48)*256
348 GOTO 356
349 REM PUSH LITERAL BYTE ON STACK
350 L=L+1
351 GOSUB 410
352 IF I<48 GOTO 334
353 IF I>57 GOTO 334
354 INPUT K
355 K=K+2304
356 GOSUB 440
357 PRINT ";";
358 GOTO 214
359 REM RELATIVE BRANCHES
360 K=T
362 GOSUB 410
363 IF I=42 GOTO 365
364 IF I<65 GOTO 334
```

```
365 GOSUB 500
366 IF T=0 IF KL+31 GOTO 334
368 IF K=O+12 THEN I=I+32
369 IF I94 GOTO 391
389 K=K-64
390 GOTO 386
391 L=L+1
392 IF I=13 GOTO 334
```

```
393 IF T=I GOTO 397
394 GOSUB 450
395 K=I
396 GOTO 386
397 K=K+128
398 GOSUB 450
399 PRINT ";";
400 IF L=J+1 GOTO 214
401 GOTO 210
402 REM      ---      SUBROUTINES
403 REM ADVANCE INPUT LINE POINTER
404 GOSUB 405
405 C=USR(P,B,USR(G,B)+1)
406 RETURN
407 REM
408 REM SKIP BLANKS IN INPUT LINE
409 GOSUB 405
410 I=USR(G,USR(G,B))
411 IF I=32 GOTO 409
412 IF I>32 RETURN
413 GOTO 941
418 REM
419 REM PRINT HEX DIGITS
420 PRINT "A";
421 RETURN
422 PRINT "B";
423 RETURN
424 PRINT "C";
425 RETURN
```

```
426 PRINT "D";
427 RETURN
428 PRINT "E";
429 RETURN
430 PRINT "F";
431 RETURN
434 IF H>64 GOTO H+H+290
435 H=H-48
436 IF H>9 GOTO 400+H+H
437 PRINT H;
438 RETURN
439 REM PRINT NUMBER AS HEX
440 H=K/4096
441 IF K<0 THEN H=H-1
442 K=K-H*4096
443 IF H<0 THEN H=H+16
444 GOSUB 436
445 H=K/256
```

```
446 K=K-H*256
447 GOSUB 436
450 H=K/16
451 K=K-H*16
452 GOSUB 436
455 H=K
456 GOTO 436
458 REM
459 REM PRINT LOCATION COUNTER
460 K=A+L
461 GOSUB 440
462 PRINT " ";
463 RETURN
498 REM
499 REM LOOK UP SYMBOL IN TABLE
500 V=0
501 W=8224
502 C=USR(G,B)
503 I=USR(G,C)
504 IF I<48 GOTO 525
505 I=USR(G,C+1)
506 IF I<32 THEN I=(USR(P,C+1,32)+USR(P,C+2,13))*0+32
508 W=USR(D,C)
509 GOSUB 404
510 IF V>0 GOTO 513
511 V=W
512 GOTO 501
513 T=S
514 GOTO 518
515 I=USR(D,T+4)
516 IF V=USR(D,T) IF W=USR(D,T+2) RETURN
517 T=T+6
518 IF T42 GOTO 510
```

```
526 T=1
527 I=L
528 GOTO 405
548 REM
549 REM PUSH 2-BYTE LITERAL ONTO STACK
550 PRINT "0A;"
552 GOSUB 460
553 L=L+2
554 GOSUB 410
555 IF I<48 GOTO 334
556 IF I>57 GOTO 334
557 INPUT K
558 GOTO 356
700 REM PROGRAM END
```

```
760 PRINT
770 PRINT X;" ERRORS"
790 END
900 REM ERROR MESSAGES
901 PRINT "*DL* ";
902 X=X+1
903 GOTO 300
911 PRINT "*IE* ";
912 X=X+1
914 L=L+2
915 GOTO 214
921 PRINT "*OP* ";
922 X=X+1
923 GOTO 214
931 PRINT "*US* ";
932 X=X+1
933 GOTO 214
941 PRINT "*LE* ";
942 X=X+1
944 RETURN
999 END
```

IMPLEMENTATION NOTES

The TINY BASIC interpreter was designed by Dennis Allison as a

Recursive Descent parser. Some of the elegant simplicity of this design was lost in the addition of syntactical sugar to the language but the basic form remains. The IL is especially suited to Recursive Descent parsing of TINY BASIC because of the general recursive nature of its procedures and the simplicity of the TINY BASIC tokens. The IL language is effectively optimized for the interpretation of TINY. Experience has shown that the difficulty of adding new features to the language is all out of proportion with the nature of the features. Usually it is necessary to add additional machine language subroutines to support the new features. Often the difficulty outweighs the advantages.

Consider for example, floating point arithmetic. This is a frequently requested addition. However, to implement floating point the following problems must be overcome:

1. Variable size. While 16 bits does not allow very large numbers, it is adequate for small integers of the kind needed for games and industrial control applications, the two environments for which TINY is most suited. But meaningful floating point numbers cannot realistically fit in less than 20 bits, and 32 bits is a much more reasonable lower limit. 26 variables of four bytes each is 104 bytes, not too large to take advantage of Page 00 addressing. Without redoing the entire ML interpreter it would be necessary to put two bytes where the variables are now and the other two in the space between 00C8 and 00FB. The expression stack may prove to be too small for very complex expressions of double-length floating point variables. This would tend to limit the allowable size of the input lines, which share the same workspace with the expression stack.

2. Number-handling routines. Not only would the arithmetic routines driving the AD, SU, MP and DV opcodes need rewriting, but also all the other opcodes which work with numbers on the stack would need modification. Otherwise the program may find it difficult to execute a GOSUB to line number 1.23E2. Perhaps a simpler alternative would be to leave the existing opcodes and add the floating point routines into the gaps in the IL instruction set, including one to fix a floating point number as well as variable load and store and the print and constant conversions. There may not be enough unused opcodes to do this without sacrificing existing functions.

3. The expression evaluation code in the IL interpreter would need revision to distinguish integer and floating point requirements, and to select the appropriate opcodes.

All in all, adding floating point operations to TINY is probably feasible, though far from easy.

On the other hand, string or array operations are probably not practical within the bounds of the present system. While all variables in TINY are predefined, arrays and variable-length strings would require memory allocation and de-allocation routines, address pointers, and dimension tables. It is conceivable that this space could be taken from the unused user program memory space, either at the end of the program (by modifying the pointer in 0024-0025 hex) or underneath the GOSUB stack (by modifying the pointer in 0022-0023 hex). In the latter case the memory allocator would need to move the stack around and also modify the stack pointer and the contents of 0026-0027 hex. Making the system invulnerable to programming errors would be extremely difficult.

Enhancements which may be considerably simpler and which should perhaps be considered first are a Logical AND function (as an intrinsic) or data indirection of the type used in NIBL.

Adding an intrinsic function consists primarily of recognizing the function name within the FACTor parsing procedure, calling EXPR to evaluate each argument, then performing the evaluation. In the case of a Logical AND function a machine language routine would be necessary for the evaluation. This may be implemented in either of two ways: the existing opcode US may be incorporated into the evaluation in which the IL interpreter knows where the subroutine is; or a new opcode may be defined. The following sequence illustrates the former technique (assume the machine language AND code at location 0003):

```
    :F20    BC F30 "AND("   RECOGNIZE FUNCTION NAME
            LN 3            LOAD ADDRESS FOR USR
            JS EXPR         GET FIRST ARGUMENT
            JS ARG          GET SECOND ARGUMENT
            BC * ")"        MUST BE RIGHT PAREN
            US              GO DO IT
            RT              RETURN TO TERM.
    :F30    ...             (REST OF FACT)
```

The indirection operator "@" could be similarly handled:

```
    :STMT   BC TLET "LET@" TEST FOR INDIRECT STORE
            LN 280         YES, SET POKE ADDRESS
            JS EXPR        GET ADDRESS
            BC * "="       NEXT MUST BE EQUAL
            JS EXPR        GET VALUE
            BE *           THAT SHOULD BE LINE END
            US             STORE THE LOW BYTE
            SP             CLEAR STACK
            NX             END OF STATEMENT
    :TLET   BC GOTO "LET"  ...ETC.
```

Indirection in the fetch is also simple:

```
    :F40    BC F5 "@"      IS IT INDIRECT?
            LN 276         YES, GET PEEK ADDRESS
            JS EXPR        GET BYTE ADDRESS
            DS             (DUMMY)
```

```
               US             GO GET IT
               RT
        :F5    BC             ...ETC.
```

When adding ML subroutines it may be helpful to know where to find some of the internal pointers used by TINY. The IL program is generally placed at the end of the ML code. Its address is stored in the two bytes which precede the Cold Start code. In other words, to find the IL base address (or to change it), follow the JMP in 0100-0103 hex, and look two bytes before its destination. This is the only copy of the address, and changes here affect the whole interpreter.

The first few instructions of the Cold Start routine define the lower bounds of the user space, so if it is necessary to add code this could be modified to leave room.

The opcode address table is placed near the beginning of the ML interpreter (right after the PEEK and POKE routines). The first six addresses select the branch instructions. Most of the unused opcodes jump to the same address. Each opcode service routine is coded as a subroutine.

Some of the Page 00 memory locations which could be of interest are defined here (all addresses are in hexadecimal):

0020-0021	Start of user program space
0022-0023	End of user program space
0024-0025	End of BASIC program, SPARE added
0026-0027	Top of BASIC stack
0028-0029	Current BASIC line number
002A-002B	IL Program Counter
002C-002D	BASIC Pointer
002E-002F	Saved Pointer
0030-007F	Input line & Expression stack
0080-0081	Random Number seed
0082-00B5	Variables
00BF	Output Column counter & Tape Mode

Other important parameters such as the RUN mode flag, the expression stack pointer, and the end of input line pointer are placed in different locations depending on the versions.

The following is an assembly listing of the currently distributed version of TINY BASIC.

```
0000 ;         .  ORIGINAL TINY BASIC INTERMEDIATE INTERPRETER
0000 ;         .
0000 ;         .  EXECUTIVE INITIALIZATION
0000 ;         .
0000 ;         :STRT PC ":Q^"        COLON, X-ON
0000 243A91;
0003 ;               GL
0003 27;             SB
0004 10;             BE L0           BRANCH IF NOT EMPTY
0005 E1;             BR STRT         TRY AGAIN IF NULL LINE
0006 59;       :L0   BN STMT         TEST FOR LINE NUMBER
0007 C5;             IL              IF SO, INSERT INTO PROGRAM
0008 2A;             BR STRT         GO GET NEXT
0009 56;       :XEC  SB              SAVE POINTERS FOR RUN WITH
000A 10;             RB                CONCATENATED INPUT
000B 11;             XQ
000C 2C;       .
000D ;         .  STATEMENT EXECUTOR
000D ;         .
000D ;         :STMT BC GOTO "LET"
000D 8B4C45D4;
0011 ;               BV *            MUST BE A VARIABLE NAME
0011 A0;             BC * "="
0012 80BD;     :LET  JS EXPR         GO GET EXPRESSION
0014 30BC;           BE *            IF STATEMENT END,
0016 E0;             SV                STORE RESULT
0017 13;             NX
0018 1D;       .
0019 ;         :GOTO BC PRNT "GO"
0019 9447CF;
001C ;               BC GOSB "TO"
001C 8854CF;
001F ;               JS EXPR         GET LINE NUMBER
001F 30BC;           BE *
0021 E0;             SB              (DO THIS FOR STARTING)
```

```
0022 10;    32       RB
0023 11;    33       GO              GO THERE
0024 16;    34 .
0025 ;      35 :GOSB BC * "SUB"      NO OTHER WORD BEGINS "GO..."
0025 805355C2;
0029 ;      36       JS EXPR
0029 30BC;  37       BE *
002B E0;    38       GS
002C 14;    39       GO
002D 16;    40 .
002E ;      41 :PRNT BC SKIP "PR"
002E 9050D2;
0031 ;      42       BC P0 "INT"     OPTIONALLY OMIT "INT"
0031 83494ED4;
0035 ;      43 :P0   BE P3
0035 E5;    44       BR P6           IF DONE, GO TO END
0036 71;    45 :P1   BC P4 ";"
0037 88BB;  46 :P2   BE P3
0039 E1;    47       NX              NO CRLF IF ENDED BY ; OR ,
003A 1D;    48 :P3   BC P7 '"'
```

36

```
003B 8FA2;  49       PQ              QUOTE MARKS STRING
003D 21;    50       BR P1           GO CHECK DELIMITER
003E 58;    51 :SKIP BR IF           (ON THE WAY THRU)
003F 6F;    52 :P4   BC P5 ","
0040 83AC;  53       PT              COMMA SPACING
0042 22;    54       BR P2
0043 55;    55 :P5   BC P6 ":"
0044 83BA;  56       PC "S^"         OUTPUT X-OFF
0046 2493;  57 :P6   BE *
0048 E0;    58       NL              THEN CRLF
0049 23;    59       NX
004A 1D;    60 :P7   JS EXPR         TRY FOR AN EXPRESSION
004B 30BC;  61       PN
004D 20;    62       BR P1
004E 48;    63 .
004F ;      64 :IF   BC INPT "IF"
004F 9149C6;
0052 ;      65       JS EXPR
0052 30BC;  66       JS RELO
0054 3134;  67       JS EXPR
0056 30BC;  68       BC I1 "THEN"    OPTIONAL NOISEWORD
0058 84544845CE;
005D ;      69 :I1   CP              COMPARE SKIPS NEXT IF TRUE
005D 1C;    70       NX              FALSE.
005E 1D;    71       J STMT          TRUE. GO PROCESS STATEMENT
005F 380D;  72 .
0061 ;      73 :INPT BC RETN "INPUT"
```

```
0061 9A494E5055D4;
0067 ;      74 :I2   BV *             GET VARIABLE
0067 A0;    75       SB               SWAP POINTERS
0068 10;    76       BE I4
0069 E7;    77 :I3   PC "? Q^"        LINE IS EMPTY; TYPE PROMPT
006A 243F2091;
006E ;      78       GL               READ INPUT LINE
006E 27;    79       BE I4            DID ANYTHING COME?
006F E1;    80       BR I3            NO, TRY AGAIN
0070 59;    81 :I4   BC I5 ","        OPTIONAL COMMA
0071 81AC;  82 :I5   JS EXPR          READ A NUMBER
0073 30BC;  83       SV               STORE INTO VARIABLE
0075 13;    84       RB               SWAP BACK
0076 11;    85       BC I6 ","        ANOTHER?
0077 82AC;  86       BR I2            YES IF COMMA
0079 4D;    87 :I6   BE *             OTHERWISE QUIT
007A E0;    88       NX
007B 1D;    89 .
007C ;      90 :RETN BC END "RETURN"
007C 895245545552CE;
0083 ;      91       BE *
0083 E0;    92       RS               RECOVER SAVED LINE
0084 15;    93       NX
0085 1D;    94 .
0086 ;      95 :END  BC LIST "END"
0086 85454EC4;
008A ;      96       BE *
008A E0;    97       WS
008B 2D;    98 .
```

37

```
008C ;      99 :LIST BC RUN "LIST"
008C 984C4953D4;
0091 ;     100       BE L2
0091 EC;   101 :L1   PC "@^@^@^@^J^@^" PUNCH LEADER
0092 24000000000A80;
0099 ;     102       LS               LIST
0099 1F;   103       PC "S^"          PUNCH X-OFF
009A 2493; 104       NL
009C 23;   105       NX
009D 1D;   106 :L2   JS EXPR          GET A LINE NUMBER
009E 30BC; 107       BE L3
00A0 E1;   108       BR L1
00A1 50;   109 :L3   BC * ","         SEPARATED BY COMMAS
00A2 80AC; 110       BR L2
00A4 59;   111 .
00A5 ;     112 :RUN  BC CLER "RUN"
00A5 855255CE;
00A9 ;     113       J XEC
```

```
00A9 380A; 114 .
00AB ;     115 :CLER BC REM "CLEAR"
00AB 86434C4541D2;
00B1 ;     116       MT
00B1 2B;   117 .
00B2 ;     118 :REM  BC DFLT "REM"
00B2 845245CD;
00B6 ;     119       NX
00B6 1D;   120 .
00B7 ;     121 :DFLT BV *          NO KEYWORD...
00B7 A0;   122       BC * "="      TRY FOR LET
00B8 80BD; 123       J LET         IT'S A GOOD BET.
00BA 3814; 124 .
00BC ;     125 .  SUBROUTINES
00BC ;     126 .
00BC ;     127 :EXPR BC E0 "-"     TRY FOR UNARY MINUS
00BC 85AD; 128       JS TERM       AHA
00BE 30D3; 129       NE
00C0 17;   130       BR E1
00C1 64;   131 :E0   BC E4 "+"     IGNORE UNARY PLUS
00C2 81AB; 132 :E4   JS TERM
00C4 30D3; 133 :E1   BC E2 "+"     TERMS SEPARATED BY PLUS
00C6 85AB; 134       JS TERM
00C8 30D3; 135       AD
00CA 18;   136       BR E1
00CB 5A;   137 :E2   BC E3 "-"     TERMS SEPARATED BY MINUS
00CC 85AD; 138       JS TERM
00CE 30D3; 139       SU
00D0 19;   140       BR E1
00D1 54;   141 :E3   RT
00D2 2F;   142 .
00D3 ;     143 :TERM JS FACT
00D3 30E2; 144 :T0   BC T1 "*"     FACTORS SEPARATED BY TIMES
00D5 85AA; 145       JS FACT
00D7 30E2; 146       MP
00D9 1A;   147       BR T0
00DA 5A;   148 :T1   BC T2 "/"     FACTORS SEPARATED BY DIVIDE
00DB 85AF; 149       JS  FACT
```

```
00DD 30E2; 150       DV
00DF 1B;   151       BR T0
00E0 54;   152 :T2   RT
00E1 2F;   153 .
00E2 ;     154 :FACT BC F0 "RND"   *RND FUNCTION*
00E2 97524EC4;
00E6 ;     155       LN 257*128    STACK POINTER FOR STORE
00E6 0A;
00E7 8080; 156       FV            THEN GET RNDM
```

```
00E9 12;   157       LN 2345         R:=R*2345+6789
00EA 0A;
00EB 0929; 158       MP
00ED 1A;   159       LN 6789
00EE 0A;
00EF 1A85; 160       AD
00F1 18;   161       SV
00F2 13;   162       LB 128          GET IT AGAIN
00F3 0980; 163       FV
00F5 12;   164       DS
00F6 0B;   165       JS FUNC         GET ARGUMENT
00F7 3130; 166       BR F1
00F9 61;   167 :F0   BR F2           (SKIPPING)
00FA 73;   168 :F1   DS
00FB 0B;   169       SX 2            PUSH TOP INTO STACK
00FC 02;   170       SX 4
00FD 04;   171       SX 2
00FE 02;   172       SX 3
00FF 03;   173       SX 5
0100 05;   174       SX 3
0101 03;   175       DV              PERFORM MOD FUNCTION
0102 1B;   176       MP
0103 1A;   177       SU
0104 19;   178       DS              PERFORM ABS FUNCTION
0105 0B;   179       LB 6
0106 0906; 180       LN 0
0108 0A;
0109 0000; 181       CP              (SKIP IF + OR 0)
010B 1C;   182       NE
010C 17;   183       RT
010D 2F;   184 :F2   BC F3 "USR"     *USR FUNCTION*
010E 8F5553D2;
0112 ;     185       BC * "("        3 ARGUMENTS POSSIBLE
0112 80A8; 186       JS EXPR         ONE REQUIRED
0114 30BC; 187       JS ARG
0116 312A; 188       JS ARG
0118 312A; 189       BC * ")"
011A 80A9; 190       US              GO DO IT
011C 2E;   191       RT
011D 2F;   192 :F3   BV F4           VARIABLE?
011E A2;   193       FV              YES.  GET IT
011F 12;   194       RT
0120 2F;   195 :F4   BN F5           NUMBER?
0121 C1;   196       RT              GOT IT.
0122 2F;   197 :F5   BC * "("        OTHERWISE MUST BE (EXPR)
0123 80A8; 198 :F6   JS EXPR
0125 30BC; 199       BC * ")"
```

```
0127 80A9;       RT
0129 2F;   .
012A ;     :ARG  BC A0 ","        COMMA?
012A 83AC;       J  EXPR          YES, GET EXPRESSION
012C 38BC; :A0   DS               NO, DUPLICATE STACK TOP
012E 0B;         RT
012F 2F;   .
0130 ;     :FUNC BC * "("
0130 80A8;       BR F6
0132 52;         RT
0133 2F;   .
0134 ;     :RELO BC R0 "="        CONVERT RELATION OPERATORS
0134 84BD;       LB 2             TO CODE BYTE ON STACK
0136 0902;       RT               =
0138 2F;   :R0   BC R4 "<"
0139 8EBC;       BC R1 "="
013B 84BD;       LB 3             <=
013D 0903;       RT
013F 2F;   :R1   BC R3 ">"
0140 84BE;       LB 5             <>
0142 0905;       RT
0144 2F;   :R3   LB 1             <
0145 0901;       RT
0147 2F;   :R4   BC * ">"
0148 80BE;       BC R5 "="
014A 84BD;       LB 6             >=
014C 0906;       RT
014E 2F;   :R5   BC R6 "<"
014F 84BC;       LB 5             ><
0151 0905;       RT
0153 2F;   :R6   LB 4             >
0154 0904;       RT
0156 2F;   .
0157 ;    0000
0000
```

INDEX

More Kim-1 and 6502 at users.telenet.be/kim1-6502.
I'm still searching for the source of this program.
Is there anyone who has access to the original papertape or ...
